

OptiCode: Machine Code Deobfuscation for Malware Analysis

Nguyen Anh Quynh, COSEINC
<aquynh -at- gmail.com>

Syscan Singapore 2013, April 25th

Agenda

- 1 Obfuscation problem in malware analysis
- 2 Deobfuscation 1: Code optimization
- 3 Deobfuscation 2: Opaque predicate
- 4 Conclusions

Malware analysis

- Analyze malware to understand what it does internally
- Focus on static analysis in this presentation

Malware analysis problems

- Understanding machine code (assembly) is hard and time-consuming
- Malware always try to make the code harder to understand for analysts
 - ▶ Use a lot of obfuscation techniques to make asm code like spaghetti
 - ▶ Even have tricks to fool reverse and analysis tools

Understanding obfuscated code

- Understand the semantics of every single instruction is always hard
- No good tool available to help analysts :-(
 - ▶ Emulator does not handle code having multiple paths
 - ▶ Nothing supports 64-bit code (x86-64 platform)

Malware obfuscation techniques

Using different tricks to obfuscate machine code

- Insert dead code 
- Substitute instruction with equivalent instructions 
- Combine all of above methods

Deobfuscate spaghetti code

Obfuscation method	Deobfuscation method
Insert dead code	?
Substitute with equivalent instructions	?

OptiCode solutions

Code optimization

- Using compiler technique to "reverse" the obfuscation techniques
- Normalize code + optimize code
- Fully automatic

Handle opaque predicate

- Furthermore remove dead code (dead branches)
- Use theorem prover (SMT solver) to decide which execution paths are infeasible
- Analysts tell OptiCode what he knows (user input required)
- Semi-automatic

Code optimization

Code optimization

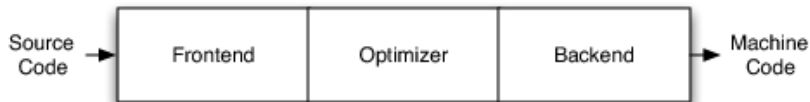
- Using compiler technique to "reverse" the obfuscation techniques
- Normalize machine code
 - ▶ Explicitly express the semantics of machine code
- Optimize normalize code
- Present optimized code as deobfuscated output

Introduction on LLVM

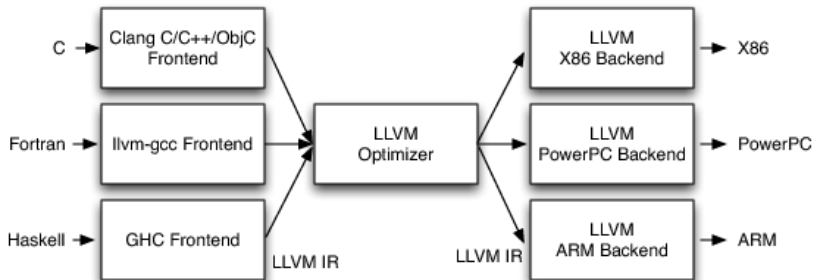
LLVM project

- Open source project on compiler: <http://www.llvm.org>
- A set of frameworks to build compiler
- LLVM Intermediate Representation (IR) with lots of optimization module ready to use

LLVM model



Compiler model



LLVM model: separate Frontend - Optimization - Backend

LLVM IR

- Independent of target architecture
- RISC-like, three addresses code
- Register-based machine, with infinite number of virtual registers
- Registers having type like high-level programming language
 - ▶ void, float, integer with arbitrary number of bits (i1, i32, i64)
- Pointers having type (to use with Load/Store)
- Support SSA by nature
- Basic blocks having single entry and single exit
- Compile from source to LLVM IR: LLVM bytecode

LLVM instructions

- 31 opcode designed to be simple, non-overlap
 - ▶ Arithmetic operations on integer and float
 - ★ *add, sub, mul, div, rem, ...*
 - ▶ Bit-wise operations
 - ★ *and, or, xor, shl, lshr, ashr*
 - ▶ Branch instructions
 - ★ Low-level control flow is unstructured, similar to assembly
 - ★ Branch target must be explicit :-(
 - ★ *ret, br, switch, ...*
 - ▶ Memory access instructions: *load, store*
 - ▶ Others
 - ★ *icmp, phi, select, call, ...*

Example of LLVM IR

```
unsigned add2(unsigned a, unsigned b) {
    if (a == 0) return b;
    return add2(a-1, b+1);
}

define i32 @add2(i32 %a, i32 %b) {
entry:
    %tmp1 = icmp eq i32 %a, 0
    br i1 %tmp1, label %done, label %recurse

recurse:
    %tmp2 = sub i32 %a, 1
    %tmp3 = add i32 %b, 1
    %tmp4 = call i32 @add2(i32 %tmp2, i32 %tmp3)
    ret i32 %tmp4

done:
    ret i32 %b
}
```

C code - LLVM IR code

Optimize LLVM bitcode

- The core components of the LLVM architecture
- Optimize performed on the bitcode (LLVM Pass) with combined/selected LLVM passes
 - ▶ Optimization to collect/visualize information
 - ▶ Optimization to transform the bitcode
 - ▶ Others
- 182 passes ready to use in LLVM 3.2

Why LLVM IR?

- Good IR for normalization phase
- Only use subset of LLVM instructions
 - ▶ Ignore instructions about high-level information from source code
- Handy frameworks to process the output IR
- Possible to optimize the LLVM bitcode resulted from the step of translating machine code → LLVM IR
 - ▶ Use suitable LLVM passes to "reverse" the obfuscation

Translate machine code to LLVM IR

- Similar to building compiler frontend for "machine code language"
- Tough due to the unstructured characteristics of machine code
 - ▶ Target of indirect branches
 - ▶ Self-modified code
- From machine code, build the Control Flow Graph (CFG) to connect basic blocks (BB)
- Translate all instructions in each BB to LLVM IR
 - ▶ Reference the ISA manual of corresponding platforms (e.x: Intel/AMD manual)



Translate x86 code to LLVM IR

and eax, ebx

```
%tmp = load i32* @_eax, align 4, !tbaa !1
%tmp1 = load i32* @_ebx, align 4, !tbaa !2
%tmp2 = and i32 %tmp1, %tmp
%tmp3 = icmp slt i32 %tmp2, 0
%tmp4 = icmp eq i32 %tmp2, 0
store i1 false, i1* @_AF, align 1, !tbaa !15
store i1 false, i1* @_OF, align 1, !tbaa !13
store i1 %tmp4, i1* @_ZF, align 1, !tbaa !9
store i1 false, i1* @_CF, align 1, !tbaa !10
store i32 %tmp2, i32* @_eax, align 4, !tbaa !1
store i1 %tmp3, i1* @_SF, align 1, !tbaa !11
```

Example of translating x86 code to LLVM IR

Optimize LLVM bitcode

- Constant propagation (-constprop)
 - ▶ $(x = 14; y = x + 8) \Rightarrow (x = 14; y = 22)$
- Eliminate dead store instructions (-dse)
 - ▶ $(y = 3; \dots; y = x + 1) \Rightarrow (\dots; y = x + 1)$
- Combine instructions (-instcombine)
 - ▶ $(y = x + 1; z = y + 2) \Rightarrow (z = x + 3)$
- Simplify CFG (-simplifycfg)
 - ▶ Remove isolated BB
 - ▶ Merges a BB into its predecessor if there is only one and the predecessor only has one successor
 - ▶ Merge a BB that only contains an unconditional branch

Deobfuscate techniques

Obfuscation method	Deobfuscation method
Insert dead code	LLVM -dse, -simplifycfg
Substitute with equivalent instructions	LLVM -constprop, -instcombine

Present deobfuscated code

- LLVM IR as output from optimization phase is already deobfuscated
- Need to present that as result: LLVM IR or assembly?
- Assembly seems good enough
 - ▶ Compile LLVM IR back to object file containing machine code
 - ▶ Disassemble machine code back to assembly
 - ▶ Original machine registers expressed as LLVM variables → memory variable

Demo

Opaque predicate

- Logical conditions cannot be evaluated with static analysis
- Typically impact conditional jumps
- Analyst can help to decide dead branches, get them removed for further deobfuscation
- Semi-automatic

Handle opaque predicate

- Create logical formula from machine code
 - ▶ Make the cuts at conditional jumps
- Combine with constraint (external knowledge) provided by malware analyst
- Feed the combined formula to **Theorem Prover** to decide if desired condition is always True, or False

Satisfiability Modulo Theories (SMT) solver

- Theorem prover based on decision procedure
- Work with logical formulas of different theories
- Suitable to express the behaviour of computer programs
- Prove the satisfiability/validity of a logical formula
- Can generate the model if satisfiable
- Prove the equivalence of two logical formulas

Z3 SMT solver

- Tools & frameworks to build applications using Z3
 - ▶ Open source project: <http://z3.codeplex.com>
 - ▶ Support Linux & Windows
 - ▶ C++, Python binding
- Support BitVector theory
 - ▶ Model arithmetic & logic operations
- Support Array theory
 - ▶ Model memory access
- Support quantifier *Exist* ($\exists$) & *ForAll* ($\forall$)

Create logical formula

Encode arithmetic and moving data instructions

Malware code

```
mov esi, 0x48  
mov edx, 0x2007
```

Logical formula

$(esi == 0x48) \text{ and } (edx == 0x2007)$

Encode control flow

Malware code

```
cmp eax, 0x32  
je $_label  
xor esi, esi  
...  
_label:  
mov ecx, edx
```

Logical formula

$(eax == 0x32 \text{ and } ecx == edx) \text{ or } (eax != 0x32 \text{ and } esi == 0)$

Create logical formula - 2

NOTE: watch out for potential conflict in logical formula

Malware code

```
mov esi, 0x48  
...  
mov esi, 0x2007
```

Logical formula

$(esi == 0x48) \text{ and } (esi == 0x2007)$

Create logical formula - 2

NOTE: watch out for potential conflict in logical formula

Malware code

```
mov esi, 0x48  
...  
mov esi, 0x2007
```

Logical formula

$(esi == 0x48) \text{ and } (esi == 0x2007)$

Malware code

```
mov esi, 0x48  
...  
mov esi, 0x2007
```

Logical formula with Single-Static-Assignment (SSA)

$(esi == 0x48) \text{ and } (esi1 == 0x2007)$

Steps to create logical formula

- Normalize machine code to LLVM IR
 - ▶ LLVM IR is simple, no overlap, no side effect (semantic explicitly)
 - ▶ LLVM IR supports Static Single-Assignment (SSA) for the step of generating logical formula
- Translate machine code to LLVM IR
- Optimize resulted LLVM bytecode
- Generate logical formula from LLVM bytecode

Generate logical formula from LLVM IR

- Perform block-by-block from the LLVM bitcode
- Generate instruction-by-instruction from LLVM IR to logical formula
 - ▶ Use theory of Boolean, BitVector or Array, depending on instruction

Logical formula for opaque predicate

- At conditional jump sites, evaluate EFlags status to find dead branches
- Example: JG depends on ZF is clear, and $SF = OF$
 - ▶ Combine with constraint ($ZF == 0 \ \&\& \ SF == OF$)
 - ▶ Lastly, combine with constraint provided by malware analyst
- Feed final formula to SMT solver to see if it is always True, or False?
 - ▶ Always True $\rightarrow$ drop fall-through branch (dead code)
 - ▶ Always False $\rightarrow$ drop target branch (dead code)
 - ▶ Neither $\rightarrow$ take both branches as usual


```
mov    edx, 0
mov    esi, 0
mov    ebx, 2
mov    eax, 14
and    esp, -0x10
mov    ecx, 29
```

Machine code

```
%tmp = load i32* @_esp, align 4, !tbaa !6
%tmp1 = and i32 %tmp, -16
store i32 %tmp1, i32* @_esp, align 4, !tbaa !6
store i32 14, i32* @_eax, align 4, !tbaa !1
store i32 0, i32* @_edx, align 4, !tbaa !4
store i32 2, i32* @_ebx, align 4, !tbaa !2
store i32 0, i32* @_esi, align 4, !tbaa !7
store i32 29, i32* @_ecx, align 4, !tbaa !3
```

LLVM IR

```
f2 = True
tmp = BitVec('tmp', 32)
f2 = And(f2, tmp == _esp)
tmp1 = BitVec('tmp1', 32)
f2 = And(f2, tmp1 == (tmp & 4294967280))
f2 = And(f2, _esp == tmp1)
f2 = And(f2, _eax == 14)
f2 = And(f2, _edx == 0)
f2 = And(f2, _ebx == 2)
f2 = And(f2, _esi == 0)
f2 = And(f2, _ecx == 29)
```

Z3 logical formula

Demo

OptiCode

- Web interface for demo
- Framework to translate x86 code to LLVM IR
- Framework to generate SMT formula from LLVM bitcode
- Support neutral disassembly engine to disassemble machine code (for normalization phase)
 - ▶ BeaEngine & Distorm are supported now
- Support 32-bit and 64-bit Intel
- Use Z3 solver to process logical formulas (opaque predicate)
- Implemented in Python & C++

Limitation & future works

Limitation

- Deobfuscated code can be even more complicated
- Self-modified code
- Indirect branches in machine code
- The efficiency of SMT solver depends on the complexity of the machine code

Future works

- Solve limitations
- Support other hardware platforms: ARM (anything else?)
- Deployed as an independent toolset for malware analysts

Conclusions

- OptiCode is useful to deobfuscate machine code
 - ▶ LLVM is useful as IR to normalize and optimize code, so obfuscated code is "reversed"
 - ▶ Dead code can be removed thanks to SMT solver (opaque predicate) with the helps from malware analysts
- Will be available to public at <http://opticode.coseinc.com>

References

- LLVM project: <http://llvm.org>
- LLVM passes: <http://www.llvm.org/docs/Passes.html>
- Z3 project: <http://z3.codeplex.com>
- Weakest-precondition of unstructured programs, Mike Barnett and K. Rustan M. Leino, PASTE 2005
- The Case for Semantics-Based Methods in Reverse Engineering, Rolf Rolles, Recon 2012

Questions and answers

OptiCode: Machine Code Deobfuscation for Malware Analysis

Nguyen Anh Quynh <aquynh -at- gmail.com>

Insert dead code

- Insert dead instruction
- Insert NOP semantic instructions
- Insert unreachable code
- Insert branch insn to next insn

Substitute instruction with equivalent instructions

Original code

```
mov esi, 0x0  
mov edx, 0x12340000
```

Obfuscated code

```
mov esi, 0x1  
dec esi  
mov edx, 0x12347891  
xor dx, dx
```



Insert dead instruction

Original code

```
...  
mov edx, 0x5555
```

Obfuscated code

```
mov edx, 0x30  
...  
mov edx, 0x5555
```

Insert NOP semantic code

Original code

```
mov edx, 0x2013
```

Obfuscated code

```
mov edi, edi  
mov edx, 0x2013  
xchg cx, cx
```


Insert branch insn to next insn

Original code

```
mov esi, 0x0  
mov edx, 0x12340000
```

Obfuscated code

```
mov esi, 0x0  
jmp $_label  
__label:  
mov edx, 0x12340000
```

